

edmroot README

Marc Paterno
CD/CEPA/APS/SLD
v00-04-02

December 19, 2003

Abstract

edmroot provides utility classes for the writing and reading of Root files containing instances of DØ chunks.

This document contains instructions for chunk developers and maintainers. It describes the modifications a chunk might need to be converted from the form suitable for use with DOOM to one suitable also for use with Root. It also describes the additional support code which chunk developers and maintainers need to write, to enable use of their chunks within Root.

Contents

1 Overview	3
2 Changes to Chunk Classes	3
2.1 Dealing with transient data members	3
2.2 Dealing with flaws in rootcint	4
2.2.1 Interaction of templates and namespaces	4
2.2.2 Template member functions	5
3 Creating Root Dictionaries	6
3.1 Directing rootcint to Build a Dictionary	6
3.2 Dealing with a Non-standard Header File Name	7
3.3 Dealing with More than One Class in a Header	7
3.4 Dealing with Dictionary Compilation Failures	7
3.5 The ROOT_COMPONENTS File	8
3.6 Library Generation	8
3.7 Dictionary Reference Headers	8
4 Branch Objects	9
4.1 Purpose of Branch Objects	9
4.2 Requirements for Branch Object CLasses	10
4.3 Registration of Branch Object Classes	10
5 Summary Checklist	11

1 Overview

The **edmroot** package contains classes and class templates that support the saving of DØ chunk instances in Root files. It also contains classes that support use of such classes at the Root application prompt.

Because the Root persistence mechanism is different from the DØOM persistence mechanism, any chunk class which is to be saved in a Root file needs to support both mechanisms. Section 2 describes the changes which most chunk classes will need in order to support Root persistence, in addition to the DØOM persistence they already support. Section 3 describes what needs to be done to cause the DØ build system to generate a Root dictionary for a class; this needs to be done for every class which will be persisted, and every class which will be used directly from the Root prompt. Section 4 describes the additional class needed to support creation of Root files. It describes how to create a “branch object” class, which is used by the *TreeWriterPkg*¹ (please see the README file in the package **tree_writer_pkg** for a description of this class) to direct the creation of branches in the Root tree.

The package **edmroot.example** contains example code, illustrating all the points in this document.

2 Changes to Chunk Classes

2.1 Dealing with transient data members

Many chunk classes contain transient data members in addition to the persistent data members. Transient data members are generally hidden from dØcint by a preprocessor directive, similar to the Figure 1.

```
#ifndef __CINT__
SomeClass  thing;
#endif
```

Figure 1: Standard DØOM hiding of transient data members.

For use with Root, such classes need to be modified so that rootcint will see the data member, but is told that the data member is not to be persisted, while keeping the data member hidden from dØcint. The modification is to use the preprocessor symbol `__DØCINT__` instead of `__CINT__`, and to tell rootcint the data member is not to be persisted by using rootcint’s “//!” preprocessor directive, as shown in Figure 2.

Please see the rootcint documentation for more details about the meaning of “//!”.

¹This is the class for the framework package responsible for writing chunks to Root files.

```

#ifndef __DOOCINT__
SomeClass    thing; //!
#endif

```

Figure 2: Modified file, hiding transient data member from DOOM and making it non-persistent in ROOT.

2.2 Dealing with flaws in rootcint

There are some features of the C++ language with which rootcint is unable to cope. This section presents those which have caused problems; it will be updated as new problems are discovered, or as new versions of rootcint remove previous failures.

The package **edmroot.example** contains the class *SampleChunk*, which illustrates how to support transient data members for both DOOM and Root.

2.2.1 Interaction of templates and namespaces

rootcint does not always deal correctly with name lookup. The most important place where this flaw causes failures is in name lookup for the types used as template arguments. Figure 3 shows code which rootcint does not handle correctly.

```

#include <vector>
#include "edm/ChunkID.hpp"
namespace edm
{
    struct MyChunk
    {
        std::vector<ChunkID> _x; // rootcint fails
    };
}

```

Figure 3: Root fails to deal correctly with name lookup.

rootcint is unable to associate the name *ChunkID* with *edm::ChunkID*, even though it is being used in a scope where the rules of C++ require this association. The work-around is to use a qualified name, even where C++ does not require it, as shown in Figure 4.

The class *SampleChunk* in package **edmroot.example** demonstrates this work-around.

```

#include <vector>
#include "edm/ChunkID.hpp"
namespace edm
{
    struct MyChunk
    {
        std::vector<edm::ChunkID> _x; // rootcint OK
    };
}

```

Figure 4: Workaround for name lookup failure in rootcint.

2.2.2 Template member functions

rootcint is generally unable to deal with member function templates. Member templates will not be usable from the Root application prompt. Usually, template members will have to be hidden from rootcint, so that it does not produce spurious warnings or errors. Figure 5 shows an example of a member function template for a simple class.

```

class SomeClass
{
public:
    template <class T> void f(T t); // rootcint fails
};

```

Figure 5: A simple class with a member template.

Member templates should be hidden from rootcint, as shown in Figure 6.

```

struct SomeClass
{
#ifdef __ROOTCINT__
    template <class T> void f(T t); // rootcint OK
#endif
};

```

Figure 6: Member templates should be hidden from rootcint.

The class *SampleChunk* in package **edmroot_example** demonstrates this work-around.

```

// contents of edm/src/TestChunk_linkdef.h
#ifdef __ROOTCINT__
#pragma link C++ nestedclass;
#pragma link C++ nestedtypedef;

#include "edmroot/ChunkWrapper.hpp"

#pragma extra_include "edmroot/ChunkWrapper.hpp"
#pragma extra_include "identifiers/EnvID.hpp"

#pragma link C++ namespace edm;
#pragma link C++ class edm::TestChunk+;
#pragma link C++ class edm::ChunkWrapper<edm::TestChunk>-;

#endif

```

Figure 7: Sample linkdef file for the class *TestChunk*.

3 Creating Root Dictionaries

For each class that will be saved to a Root file, and which will be used from the Root prompt, it is necessary to create a Root dictionary. This includes not only chunk classes themselves, but also the classes of persistent data members of chunks, and any other class used in the interface of a chunk, and for which the chunk author wants to provide functionality at the Root application prompt.

3.1 Directing rootcint to Build a Dictionary

In the standard DØ file naming scheme, a class *SomeClass* will be defined in a the header *SomeClass.hpp*. In order to have the DØ build system create the Root dictionary for this class, the chunk maintainer must put the appropriate “linkdef” file in the source directory for the package.

Figure 7 shows the linkdef file for the class *edm::TestChunk*. The header for this class is *edm/edm/TestChunk.hpp*, so the linkdef file is called *edm/src/TestChunk_linkdef.h*.

This linkdef file contains directives to generate a dictionary for each of two different classes: the chunk class (*edm::TestChunk*) and also an **edmroot** support class, *edm::ChunkWrapper<edm::TestChunk>*.

Please note the + or – after the classnames. These are not typographical errors, and are required. See the rootcint documentation for a full explanation.

The file *src/SampleChunk_linkdef.h* in package **edmroot_example**

demonstrates what is needed to support a standard chunk.

3.2 Dealing with a Non-standard Header File Name

Some packages may use other file extensions (for example, `.h`) other than `.hpp` for class headers. The presence of a `linkdef` file (described in section 3.1 and the mentioning of the class in the `ROOT_COMPONENTS` files (described in section 3.5 will cause the generation of a dictionary which expects a header with the extension `.hpp`. In such a case, the simplest solution is to add an additional header, following the standard naming scheme, to the package's header directory. This header need only include the header with the non-standard naming scheme. No other entry is needed. Figure 8 shows the file `Example.hpp`, which includes the non-standard named `Example.h`.

```
#include "example_package/Example.h"
```

Figure 8: Example of handling a non-standard header name.

Note that not even an include guard is needed, since the header `Example.h` must already contain one.

3.3 Dealing with More than One Class in a Header

In some cases, there may be more than one class defined in a given header. In such a case, the corresponding `linkdef` file should contain a “`pragma link C++ class`” directive for each class to be used by Root, and a “`pragma link C++ namespace`” directive for each namespace in the file. Make sure to include nested classes.

The file `src/SampleChunk.linkdef.h` in package **edmroot_example** demonstrates how to deal with additional classes defined in one header.

3.4 Dealing with Dictionary Compilation Failures

In some cases, there is a need for additional headers in the dictionary file. This most often happens when forward declarations are used in the header being processed; in order to generate the code for interactive use at the Root prompt, a forward declaration is often insufficient. In such cases, one might need to add extra “include” directives in the `linkdef` file; it may also be necessary to use “`pragma extra_include`”. Figure 9 shows an example, as appears in the header `edm/src/AbsChunk.linkdef.h`.

In this case, the header for `edm/edm/AbsChunk.hpp` forward declares the names `edm::RCPID` and `edm::EnvID`, which is sufficient for the use it makes of those names. However, the Root dictionary contains

```
// contents of edm/src/AbsChunk_linkdef.h
#ifdef __ROOTCINT__

#pragma link C++ nestedclass;
#pragma link C++ nestedtypedef;

#pragma link C++ namespace edm;
#pragma link C++ class edm::AbsChunk+;

// The dictionary needs these headers so that some
// member functions can be wrapped, but we don't want
// them in the AbsChunk.hpp header file.
#pragma extra_include "identifiers/RCPID.hpp"
#pragma extra_include "identifiers/EnvID.hpp"
```

Figure 9: Example of the use of include directives in a linkdef file.

code which makes additional use of these types, and so requires a the inclusion of the headers defining them.

In some cases, it is also necessary to add an include directive to the linkdef file. Please see the rootcint documentation for a full explanation of linkdef files.

The file `src/SampleChunk_linkdef.h` in package **edmroot_example** demonstrates to use the “extra_include” directive to cure dictionary compilation failures.

3.5 The ROOT_COMPONENTS File

Packages that use the **ctbuild** build system must include a `ROOT_COMPONENTS` file, similar in nature to the `DOOM_COMPONENTS` file. Only those files mentioned in the `ROOT_COMPONENTS` file will be processed by rootcint; the existence of a linkdef file is not sufficient.

3.6 Library Generation

The DØ build system will automatically place the generated Root dictionaries for each package into the same library as the rest of the package. This means that each package for which a Root library is generated will have a link dependence on Root.

3.7 Dictionary Reference Headers

Framework programs that use *TreeWriterPkg* to write files need to make use of the Root dictionaries for each of the classes to be written. How-

ever, the organization of the code is such that no link-time dependence on the libraries containing the dictionaries is present. To help cause such a dependence when necessary, each linkdef file must be accompanied by a *dictionary reference header*.²

For each chunk, the chunk maintainer must provide a header file `X_dict_ref.hpp`. In this file, one must invoke the macro `GETDICT`, using the name of the chunk class (with any namespace qualification) as the argument for the macro. One must also `#include` the dictionary reference headers for whatever classes the chunk class depends upon. One must also `#include` `edmroot/macros.hpp`, where `GETDCIT` is defined. Figure 10 shows an example, for the class `TestChunk3`.

```
#ifndef EDMROOT_TESTCHUNK3_DICT_REF_HPP
#define EDMROOT_TESTCHUNK3_DICT_REF_HPP

#include "edmroot/macros.hpp"
#include "edmroot/ChunkWrapperBase_dict_ref.hpp"

GETDICT(TestChunk3);

#endif // EDMROOT_TESTCHUNK3_DICT_REF_HPP
```

Figure 10: An example of a dictionary reference file.

4 Branch Objects

4.1 Purpose of Branch Objects

In order to allow a user to decide which chunk instance will be written to which branch of the Root tree, each chunk developer and maintainer is expected to write one (or more) “branch object” class(es).

The purpose of a “branch object” is to:

- create the branch on which instances of the associated class will be saved,
- select the correct *single* instance of the associated chunk class, and
- give that chunk to the `edm::TreeWriter` to be saved in the Root output file.

²See the **tree.writer.package** README file for a description of the use of dictionary reference headers.

4.2 Requirements for Branch Object Classes

A branch object class must meet the following requirements:

- It must inherit from *d0root::BaseBranchObject*.
- It must have a constructor which takes three arguments. The first argument, of type `const std::string&`, is the name of the branch to which the chunks will be written. The second argument, of type `const std::vector<std::string>&`, contains (in string form) parameters which can be used in any fashion the branch object author wishes; these parameters are set in the RCP which constructs the *TreeWriterPkg* object, as described in the README file for **tree_writer_pkg**. The third argument, of type `const edm::RCP&`, is the entire package RCP used to configure the *TreeWriterPkg* instance.
- It must have a member function `makeBranch(edm::TreeWriter&)` which must call `TreeWriter::makeBranch<X>`, using the associated chunk type as *X*.
- It must have a member function `fill(edm::TreeWriter&, const edm::Event&)` which obtains the correct chunk from the given *edm::Event*, and inserts it into the given *edm::TreeWriter*.

4.3 Registration of Branch Object Classes

A registration mechanism is used to announce branch object classes to a factory which is responsible at runtime for making the branch object instances. To support this registration mechanism, the chunk author must make use to two macros: `REG_BBO_DECL` and `REG_BBO_IMPL`, defined in the header `edmroot/RegistryMacros.hpp`.

The branch object class associated with a chunk class *X* should be *XBO*, and should be defined in a header named *XBO.hpp*. It should be accompanied by a “registration header” *XBO_ref.hpp*. The macro `REG_BBO_DECL` should be invoked in this header. Figure 11 shows an example branch object registration header, for the class *ns::XChunkBO*.

The companion macro `REG_BBO_IMPL` should be invoked in the implementation (*.cpp*) file for the branch object class. It also takes two arguments, the namespace and the classname for the branch object which it is to register.

Note that this macros can not deal with nested namespaces; branch object classes should not be declared in a nested namespace. This restriction may be lifted in a later version.

```

#ifndef EXAMPLE_XCHUNKBO_REF_HPP
#define EXAMPLE_XCHUNKBO_REF_HPP

#include "edmroot/RegistryMacros.hpp"

REG_BBO_DECL(ns, XChunkBO);

#endif // EXAMPLE_XCHUNKBO_REF_HPP

```

Figure 11: The branch object registration file for class *ns::XChunkBO*.

5 Summary Checklist

The following is a summary check-list of the steps needed to support a chunk for use in Root.

- Handle transient data members for Root, as well as for D00M.
- Hide template members from rootcint.
- Write linkdef files for files to be processed by rootcint.
- Write a dictionary reference file for each chunk.
- Write a “branch object” that others will use to direct the selection of which chunk instance will be saved on which branch, and a “branch object registry” file to direct program linking. Remember to invoke the REG_BBO_IMPL macro in the implementation file for the branch object.
- Write a branch registry header for the branch object, which invokes the REG_BBO_DECL macro for the branch object class.

Finally, it is important to write and run tests to make sure all is working correctly. These tests should include creation of one or more chunk(s), saving to and restoring from a D00M format file, saving to and restoring from a Root file, and manipulation of the restored objects from the Root prompt. There are many kinds of defects in the construction and use of the Root dictionaries which the compiler can not diagnose. As a result, testing of the system is crucial.